

Singleton SimObjects in the Torque Game Engine

Steven Peterson

February 28, 2006
Revised: March 30, 2008

Abstract

Introduction

Singleton Classes are a C++ Design Pattern that can help keep your programs clean and efficient. This is done by enforcing a maximum of one instance of the designated class which is only instantiated on-demand, and keeping it in the global scope for reuse until program-exit, rather than destroying and re-creating it multiple times.

`SimObject` is the abstract base-class from which all assets in the *Torque Game Engine (TGE)* simulation are derived. Understanding how to wield `SimObjects` and their associated classes is a core concept, key to working with TGE effectively.

Combining these two concepts is not stright forward, as the traditional example of one breaks the other. This article begins with a brief introduction to each concept since both are underdocumented in the Torque Community, and then discusses several possible strategies for combining the two.

Prerequisites

This article assumes that you are familiar with C++ and the basics of the Object Oriented Paradigme. It assumes you are comfortable editing the Torque Source Code and recompiling it on your own. You need not have prior experience with SimObjects or Design Patterns to benifit from this article.

Contents

1	About SimObjects	3
1.1	What are SimObjects?	3
1.2	Why Use SimObjects?	3
1.3	Template SimObject Class	4
1.4	Final Notes on SimObjects	6
2	About the Singleton Pattern	7
2.1	What is the Singleton-Pattern?	7
2.2	How is it accomplished?	7
2.3	Further Reading	7
2.4	Double Locking Pattern	7
2.5	Classic Singleton Implementation	8
3	The Singleton-SimObject Contradiction	10
3.1	Problems Arise	10
3.2	Solving the Puzzle	11

1 About SimObjects

1.1 What are SimObjects?

SimObjects are one of the most powerful and possibly least understood classes in the Torque Source Code. That said, SimObjects are NOT hard to learn! They are not some mystical form of Voodoo and you need not avoid them like the plague.

SimObjects are derived from base-class 'ConsoleObject'. Inheriting from ConsoleObject is what makes a C++ class visible to Torque-Script and the Torque-Console. However don't rush to subclass ConsoleObject directly! The entire purpose of SimObject is to give some extra functionality for the same amount of work, to classes that have to sub-class ConsoleObject anyway.

So to answer the question: SimObjects are simply C++ classes(in Torque) derived from the 'SimObject' base-class, which is in turn, derived from the 'ConsoleObject' base-class.

1.2 Why Use SimObjects?

Good Question. Aside from being visible to the script/console SimObjects have another powerful feature. With a few simple lines of code, an instance of your SimObject sub-class can be 'registered' with Torque by an id# AND a name. This means you don't have to keep as careful track of the instance as you can always find it again.

Not only that but you can (you don't have to) override two SimObject functions called `onAdd()` and `onRemove()` which are called when the object is registered or unregistered from the namespace.

Finally SimObjects are easy to create and work with once you learn how. Below are some links to documentation that will help you on your way. For now just follow the rest of this tutorial and you will get the basic idea.

1.3 Template SimObject Class

```
#ifndef _MYOBJECT01_H_
#define _MYOBJECT01_H_
/** @file *****/
// filename      : MyObject01.h
// date          : February 28th, 2006
// author        : Steven Peterson : info _A stevengpeterson _D com
// *****/
#include "console/simBase.h"

/**
 * @section MyObject01_intro Introduction
 * This is an example subclass of SimObject. You can use it
 * as a template for creating your own SimObject's!
 */
class MyObject01 : public SimObject {
public:
// Special Declarations
    DECLARE_CONOBJECT( MyObject01 );    // Important!

// Derived Methods
    static void initPersistFields();    // Optional Method
    virtual bool onAdd();                // Optional Method
    virtual void onRemove();            // Optional Method

private:
// Special Declarations
    typedef SimObject Parent;          // Important!

// Private Members
    S32 mValue1;
    S32 mValue2;
};
#endif
```

Some things to note before moving on:

1. class TestClass1 : public SimObject {
2. DECLARE_CONOBJECT(TestClass1);
3. typedef SimObject Parent;

First we use inheritance to derive from base-class SimObject. Simple enough. Second we add the DECLARE_CONOBJECT macro to the public section. This is just needed by SimObject, along with the matching IMPLEMENT_CONOBJECT macro in the next section. Lastly we see the typedef in the private section. This lets the script-console know who the parent class is. We also use it to refer to the parent (SimObject) in the source-section below.

The three public methods are all inherited from SimObject and overriding them is optional. There are others also, but they are beyond this articles scope.

```

/** @file ****
// filename   : MyObject01.cc
// date      : February 28th, 2006
// author    : Steven Peterson : info _A stevenpeterson _D com
// ****
#include "console/consoleTypes.h"
#include "console/simBase.h"
#include "MyObject01.h"

// Macros and Statics:
IMPLEMENT_CONOBJECT( MyObject01 );      // Important!

/**
 * Makes (some) class members visible to the script-side.
 */
void MyObject01::initPersistFields() {
    Parent::initPersistFields(); // adds the parent class's fields as well.

    addGroup("Misc");
        addField("value1", TypeS32, Offset(mValue1, TestClass1));
        addField("value2", TypeS32, Offset(mValue1, TestClass1));
    endGroup("Misc");
} // end function: initPersistFields()

/**
 * Called when object is registered with the simulation.
 */
bool MyObject01::onAdd() {
    // Calls the parents 'onAdd()' - Important!
    if(!Parent::onAdd())
        return false;

    // print message to the console
    Con::printf("TestClass1 Successfully Registered");
    return true;
}

/**
 * Called when object is UNregistered from the simulation.
 */
void MyObject01::onRemove() {
    // Calls the parents 'onRemove()' - Important!
    Parent::onRemove();

    // print message to the console
    Con::printf("TestClass1 Successfully UnRegistered");
}

```

The most important thing here is to get the `IMPLEMENT_CONOBJECT` at the top. As I will come back to presently, this is important for all Console-Objects, and therefore SimObjects.

Next is a standard example of `InitPersistFields`. This is what makes your variables visible to script. Having done this you could add your `TestClass1` to the `SimGroup(MissionGroup)` in your `*.mis` script-file, the same as all the other simobjects there. You would then be able to edit the `value1` and `value2` items using the in-game editor.

For `onAdd()` and `onRemove()` calling the parent is required. The `con::printf` line is optional but sends a useful message to the console and console log.

1.4 Final Notes on SimObjects

It is possible to register your SimObject with the simulation through C++ code. If you create the object in Torque-Script this is done for you automatically. More will be said about all of this as we go along. Congratulations you just passed the SimObjects crash-course!

2 About the Singleton Pattern

2.1 What is the Singleton-Pattern?

Sometimes there is a class that you only want one instance of in your program. A common reason for this may be a class that manages other objects or events in the program. Or perhaps there is a sub-class you would otherwise have to create and destroy each time you needed. The purpose of the Singleton-Pattern is to:

“Ensure a class only has one instance, and provide a global point of access to it.” (Gamma, Helm, Johnson, Vlissides)

2.2 How is it accomplished?

Simply, but not intuitively! The trick is to place the class constructor in the **private** section of the class!

Also in the private section, declare a **static pointer** of the class' type. Marking the variable static means there will only be one copy of it in memory whether you have zero, one, or 20 instances of the class; they all point to the same memory location.

Last we make a **public static** method called `getInstance()`. Like before the **static** declaration works similarly in this case, enabling us to call the function, even if the class has not yet been instantiated!

2.3 Further Reading

- Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Mass.: Addison-Wesley, 1995.
- Shalloway, A., Trott, J., Design Patterns Explained: A New Perspective on Object-Oriented Design, Boston, Mass. Addison-Wesley, 2002.

Note that the second book may provide an easier introduction for beginners to the topic of design-patterns.

2.4 Double Locking Pattern

For those who are interested, a similar pattern known as the *“Double Locking Pattern”* provides the same functionality as the Singleton Pattern, in a multi-threaded environment.

2.5 Classic Singleton Implementation

```
#ifndef _SINGLETON01_H_
#define _SINGLETON01_H_
/** @file *****
// filename      : Singleton01.h
// date          : February 28th, 2006
// author        : Steven Peterson : info _A steven@peterson _D com
// *****/

#include "console/simBase.h"

/**
 * @section Singleton01_intro Introduction
 * This is an example singleton-pattern class. You can use this
 * as a template for creating your own singleton classes.
 */
class Singleton01 {
public:
    // Public Methods
    static Singleton01* getInstance();

private:
    // Private Methods
    Singleton01() {};                // Default Constructor

    // Private Members
    static Singleton01 *sInstance;
};
#endif
```

Here we can clearly see the:

- public static getInstance()
- private constructor
- private static *sInstance

That's all one needs to create a singleton normally. Next we will see the implementation is just as simple!

```

/** @file *****
// filename      : Singleton01.cc
// date         : February 28th, 2006
// author       : Steven Peterson : info _A stevenpeterson _D com
// *****/

#include "console/consoleTypes.h"
#include "console/simBase.h"
#include "testClass2.h"

// initialize static variable 'sInstance' to NULL
Singleton01* Singleton01::sInstance = 0;

//-----
// Public Methods:
//-----

Singleton01* Singleton01::getInstance() {
    if (sInstance == 0)           // test to see if the object exists
        sInstance = new Singleton01; // if not Create it

    // return the (only) class instance
    return sInstance;
}

```

Notice in the 'Macros and Statics' section I initialized *sInstance to NULL outside of any function. This is legal because the variable is static, and the line is executed on start-up before "int main" begins executing.

Next take a look at the implementation of getInstance(). The object constructor is only called if needed, and then a pointer to the object is always returned.

And finally that is how you call getInstance() from anywhere else in your code:

```

#include "Singleton01.h"
...
...
Singleton01* mySingleton = Singleton01::getInstance();

```

3 The Singleton-SimObject Contradiction

3.1 Problems Arise

There are a lot of good reasons for wanting to combine the Singleton-Pattern with a SimObject subclass. Unfortunately deriving from "ConsoleObject" (and therefore, SimObject) requires the constructor to be public! Why is this you ask?

The reason the constructor can't be private is that on Torque-Startup, ConsoleObject initializes and then immediately destroys all its subclasses to get information about them. To my understanding it is checking all of the `initPersistFields`. I also believe this is why the macros `DECLARE_CONOBJECT` and `IMPLEMENT_CONOBJECT` are needed. You should keep this in mind when writing your constructor as it can be the source of annoying bugs and crashes on startup.

Well, that pretty much kills every thing we just learned; right? There are a few work-arounds. While none of the following examples are quite as elegant as the original Singleton-Pattern model, they get the job done.

3.2 Solving the Puzzle

The two major properties of a singleton class are *single-instance* and *lazy-instantiation*. We can still implement both of these in Torque, if less elegantly than before.

This setup assumes the class will be called from script which registers it in the simulation automatically or registered in C++ at some other point. Then you would be able to find the object through the Sim::namespace.

The idea here is to simply move singleton constructor back to `public`, and assert that there is no other existing instances of it. We still create a `getInstance()` and tell people to use it. If the `assert(...)` fails, it means somebody called the constructor directly after an instance of the class had already been created.

Remember that `assert`'s go away in a *Release* compile, so this is less than full-proof, but if your code is carefully tested in debug-mode you should be ok.

```
#ifndef _SING_SIMOBJECT_01_H_
#define _SING_SIMOBJECT_01_H_
/** @file *****
// filename      : SingSimObject01.h
// date          : February 28th, 2006
// author        : Steven Peterson : info _A stevenpeterson _D com
// *****/

#include "console/simBase.h"

/**
 * @section SingSimObject01_intro Introduction
 * This is an example subclass of SimObject. You can use it
 * as a template for creating your own SimObject's!
 */
class SingSimObject01 : public SimObject {
public:
// Special Declarations
    DECLARE_CONOBJECT( SingSimObject01 );    // Important!

    SingSimObject01();                      // Default Constructor
    ~SingSimObject01();                      // Default Destructor

    static SingSimObject01* getInstance();

// Derived Methods
    static void initPersistFields();        // Optional Method
    virtual bool onAdd();                   // Optional Method
    virtual void onRemove();                // Optional Method

private:
// Special Declarations
    typedef SimObject Parent;              // Important!

// Private Members
    static SingSimObject01 *sInstance;     // Pointer to Singleton-Instance
};
#endif
```

```

/** @file *****/
// filename    : SingSimObject01.cc
// date       : February 28th, 2006
// author     : Steven Peterson : info _A steven.peterson _D com
// *****/

#include <cassert>                // Needed for 'assert'
#include <cstdlib>                // Needed for 'assert'
#include "console/consoleTypes.h" // Needed for InitPersit 'Type' fields
#include "console/simBase.h"
#include "SingSimObject01.h"

// Macros and Statics:
IMPLEMENT_CONOBJECT( SingSimObject01 ); // Important!

// initialize static variable 'sInstance' to NULL
SingSimObject01* SingSimObject01::sInstance = 0;

/**
 * Default Constructor
 */
SingSimObject01::SingSimObject01() {
    // Singleton Assertion
    assert( sInstance == 0 );
    sInstance = this;
}

/**
 * Returns a pointer to the SingSimObject01 instance.
 */
SingSimObject01* SingSimObject01::getInstance() {
    if (sInstance == 0) // test to see if the object exists
        sInstance = new SingSimObject01; // if not Create it

    // return the (only) class instance
    return sInstance;
}

/**
 * Default Destructor
 */
SingSimObject01::~SingSimObject01() {
    // Release Singleton Check
    sInstance = 0;
}

/**
 * Makes (some) class members visible to the script-side.

```

```

*/
void SingSimObject01::initPersistFields() {
    Parent::initPersistFields(); // adds the parent class's fields as well.

    addGroup("Misc");
        addField("value1", TypeS32, Offset(mValue1, TestClass3));
        addField("value2", TypeS32, Offset(mValue1, TestClass3));
    endGroup("Misc");
} // end function: initPersistFields

/**
 * Called when object is registered with the simulation.
 */
bool SingSimObject01::onAdd() {
    // Calls the parents 'onAdd()' // Important!
    if(!Parent::onAdd())
        return false;

    // print message to the console
    Con::printf("SingSimObject01 Successfully Registered");
    return true;
}

/**
 * Called when object is UNregistered from the simulation.
 */
void SingSimObject01::onRemove() {
    // Calls the parents 'onAdd()' // Important!
    Parent::onRemove();

    // print message to the console
    Con::printf("SingSimObject01 Successfully UnRegistered");
}

```